RAII and Associated Magic

Cristian Cibils
(ccibils@stanford.edu)

Classes & Inheritance

- A bit of a monster
  - Designing Types
  - Designing Templated Types
  - Const Correctness
  - Operator Overloading
  - Constructors & Assignment
  - RAII

Review: Constructors

- The **constructor** for an object transforms **uninitialized** data into valid data
- A constructor which can be called with no arguments is called a **default constructor**
- In general, constructors can take any number of arguments
- However, they do not return a value

Review: Constructors

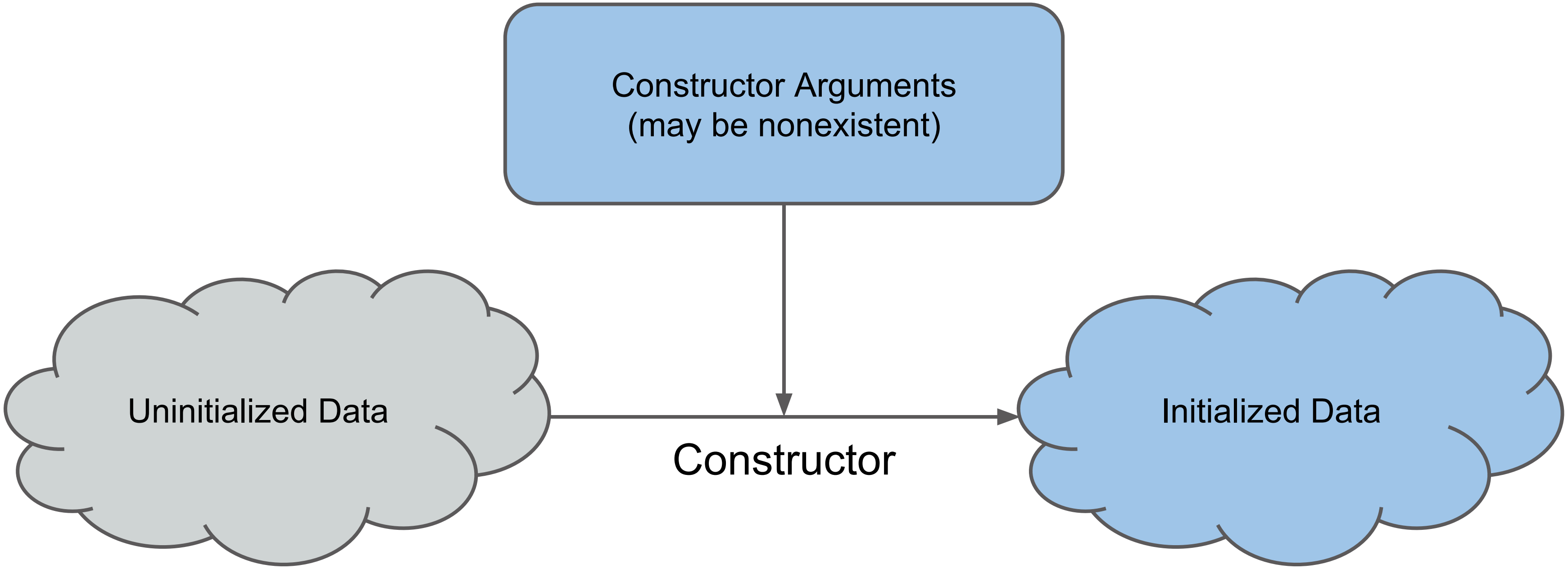
- The constructor which takes an object of the same type as an argument is called the **copy constructor**
- This constructor **initializes** junk data using an **existing object**
- The data in the newly initialized object should be the same a copy of the data in the existing object.

Review: Assignment Operator

- The form of operator= which takes an object of the same type is called the **assignment operator**
- This is used to replace **existing** data with a different bit of **existing** data

Review: Constructors

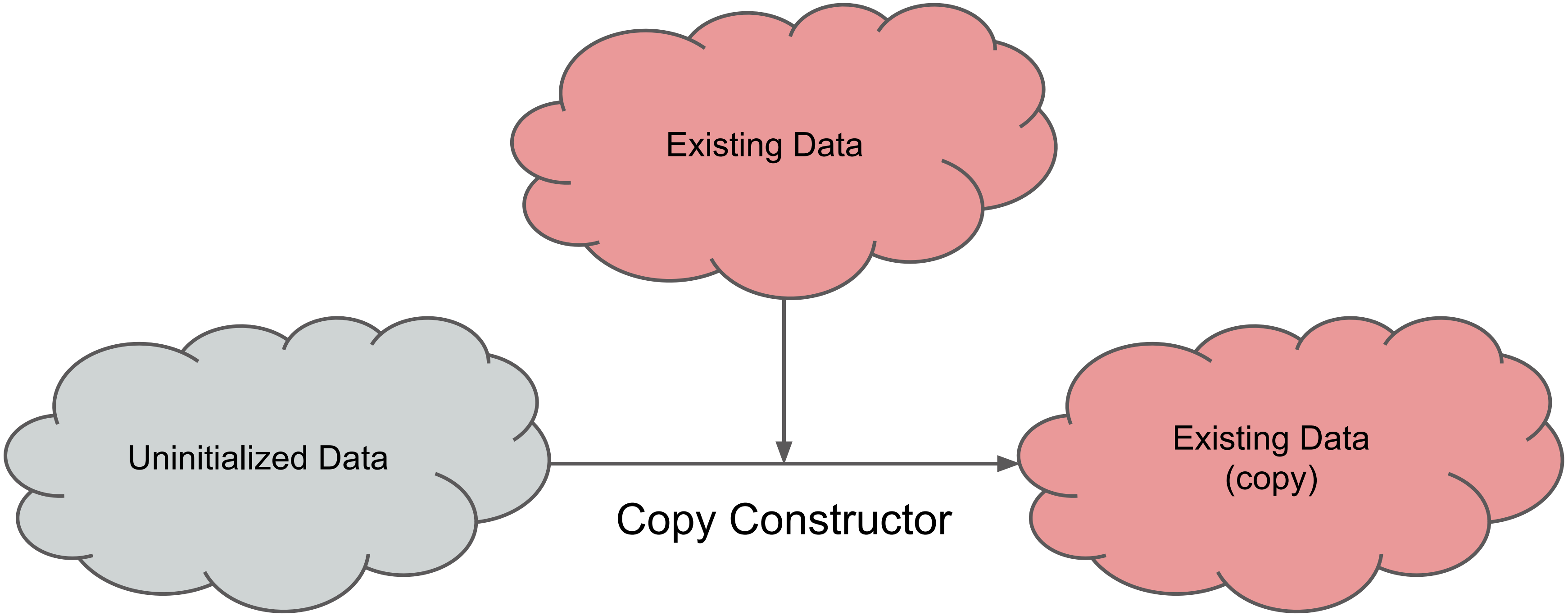
Up to now, I've discussed constructors as a means of **initializing** data, or giving member variables their starting values



```
vector<int> x;
vector<int> y(42, 10);
```

Review: Constructors

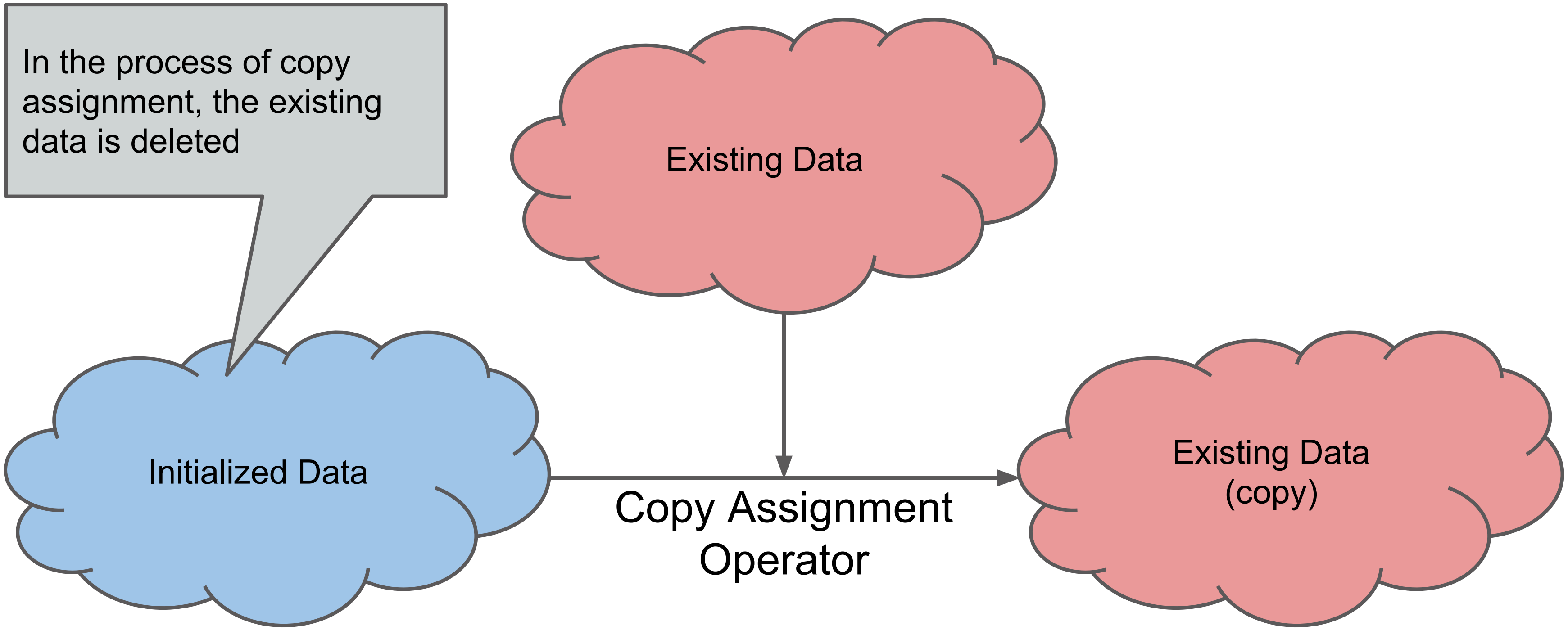
We also talked about the copy constructor, which replaced existing valid data with different valid data.



```
vector<int> x(42, 10);
vector<int> y = x;
vector<int> z(x);
```

Review: Constructors

We also talked about the copy assignment operator, which replaced existing valid data with different valid data.



In the process of copy assignment, the existing data is deleted

Existing Data

Initialized Data

Copy Assignment Operator

Existing Data (copy)

```
vector<int> x(42, 10);
vector<int> y;
y = z;
```

C File I/O

Instead of jumping into RAII, I'm first going to give a quick summary of how file processing is done in C, because it's a great way to explain RAII.

C File I/O

- To read a value from a file, you first open it with **`fopen`**
- We read data with **`fgetc`** and **`fgets`**
- We then have to close a file using **`fclose`**

C File I/O

When programmers forget to call fclose, bad things happen, from memory leaks to crashes.

Constructors: Take Two

Up until now we've been talking in terms of **initialization** -- transforming junk data into valid data.

Resources

I now want you to think of things in terms of **resources**

Resources

- What's a resource?
  - Something you have to **acquire** and **release**
  - You must acquire a resource before using it and release it when done (preferably as soon as possible).
- Let's look at a real life example of what a resource is

Resources

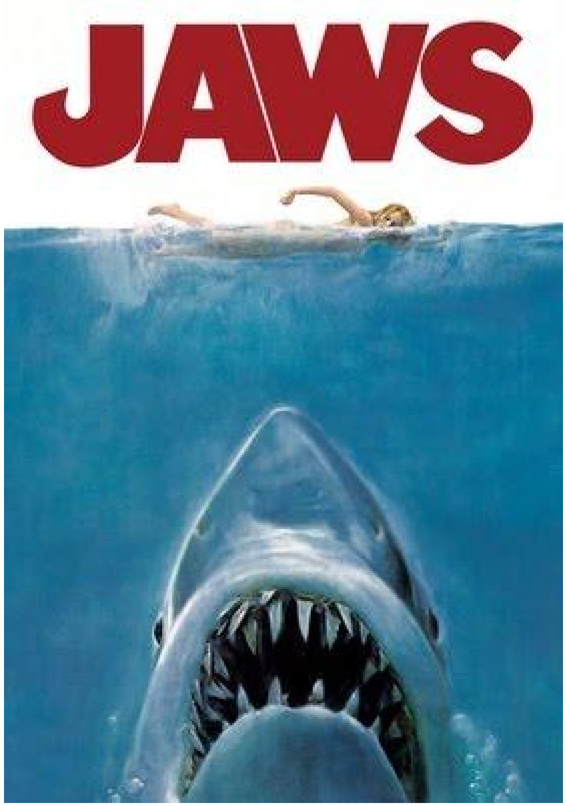
- Let's say you're a photographer trying to get pictures of sharks
- Before you go swimming, you'll need to **acquire** a shark proof cage

Resources

- With your cage, you can be safe photographing sharks
- Once done, you return the cage
- Look at the cute shark!

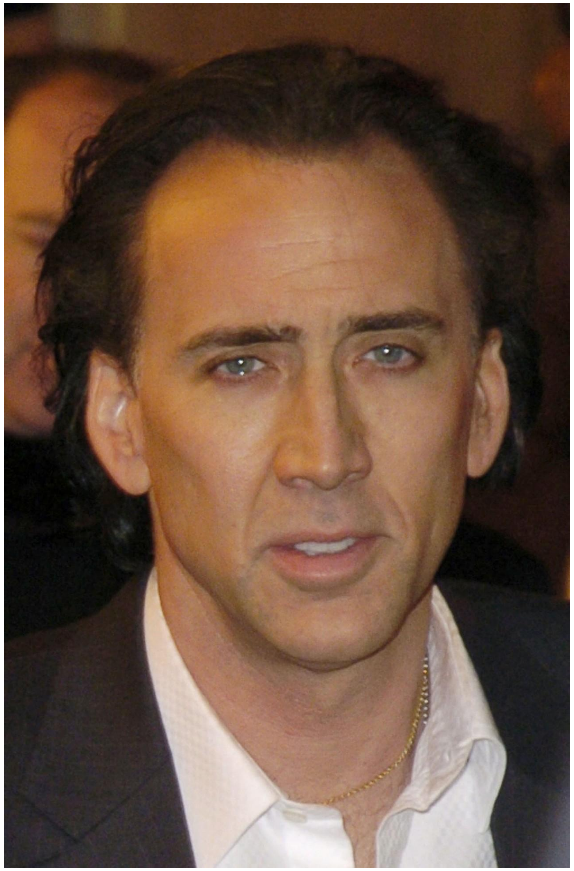
Resources

- If you relied on the resource without acquiring it, errors can occur
- In this case the error is sharks



Resources

If you forget to **release** your shark proof cage, you'll be stuck in a cage until you do



Resources

Don't worry, resources are applicable for purposes other than photographing sharks

	Acquire	Release
Files	`fopen`	`fclose`
Memory	`new, new[]`	`delete, delete[]`
Locks	`lock, try_lock`	`unlock`
Sockets	`socket`	`close`

Resources

```
// Here's C file I/O with resources marked
void printFile(const char* name) {
  // Acquire the resource
  FILE *f = fopen(name, "r");

  // Print the contents of 'f'

  // Release the resource
  fclose(f);
}
```

Resources

```
// We can forget to acquire a resource...
void printFile(const char* name) {
  FILE *f; // oops!

  // This part will probably break!

  fclose(f);
}
```

Resources

```
// We can forget to release a resource
void printFile(const char* name) {
  FILE *f = fopen(name, "r");

  // Print the contents of 'f'

  // The program will now waste memory
  // It may even crash!
}
```

Resources

What's so great about this abstraction of a resource though? Why do we care that these different concepts have this common structure?

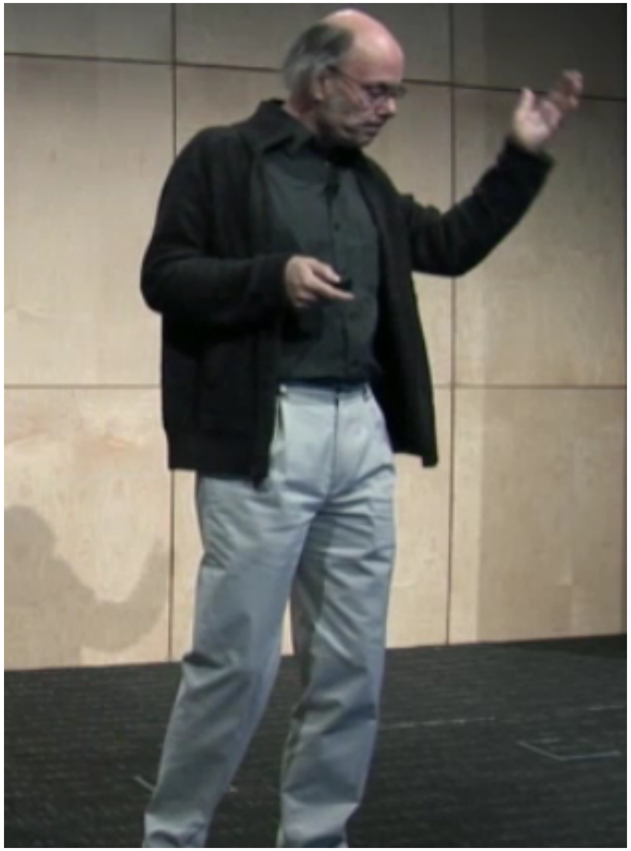
RAII

"Resource Allocation is Initialization"

RAII

The name isn't exactly great...

- *"The best example of why I shouldn't be in marketing"*
- *"I didn't have a good day when I named that"*



Bjarne Stroustrup, still unhappy with the name RAII in 2012

RAII

- Creating an object calls its constructor, acquiring the resource
  - This will happen when you declare the variable, or create it with `new`
- When an object's destructor is called the resource will be freed
  - This happens when the object goes out of scope or gets `deleted`

RAII

```
// Remember this code?
void printFile(const char* name) {
   FILE *f = fopen(name, "r");

   // Print the contents of 'f'

   fclose(f);
}
```

RAII

Let's see if the magic of RAII can help out with this...

RAII

```
struct FileObj {
   FILE *ptr;
   // Acquire the file resource
   FileObj(char *name)
      : ptr(fopen(name, "r")) {}

   // Release the file resource
   ~FileObj() {
      fclose(ptr);
   }
};
```

RAII

```
void printFile(const char* name) {
   // Initialize the object
   // Implicitly acquire the resource
   FileObj o(name);

   // Print the contents of the file

   // Destructor the object
   // Implicitly release the resource
}
```

RAII

Is that all that this does though? Just catches problems when you forget to fclose at the end of a function?

RAII

```
void printFile(const char* name) {
   FILE *f = fopen(name, "r");

   // Skip files starting with 'a'
   if (fgetc(f) == 'a')
      return;

   // Print file contents

   fclose(f);
}
```

RAII

```
void printFile(const char* name) {
   FILE *f = fopen(name, "r");

   // Skip files starting with 'a'
   if (fgetc(f) == 'a')
      return; // where's the fclose?

   // Print file contents

   fclose(f);
}
```

RAII

- You've already been using RAII!
  - You can construct an `ifstream` with a filename and it will open the file
  - When the `ifstream` gets destroyed, the destructor automatically closes the file
- There are also `.open()` and `.close()` functions, but they aren't necessary

Smart Pointers

Let's quickly take a look at another great application of RAII: smart pointers

Standard smart pointers require C++11

Smart Pointers

- Memory leaks (acquiring memory and never deleting it) are **bad**
- This team got knocked out of a $2M robot race because of memory leaks

http://www.codeproject.com/Articles/21253/If-Only-We-d-Used-ANTS-Profiler-Earlier

Smart Pointers

- Our first attempt at a RAII based pointer might work something like this:
  - Handle initialization of the pointer resource in the constructor
  - Free any associated memory when the object is destroyed
  - Allow access to the underlying pointer with operator* and operator->
  - To copy a smart pointer, copy the stored pointer value
- Let's look at a very simple example

Smart Pointers

```
void f() {
  // First, we heap allocate a string
  string *x = new string("hi!");

  cout << *x << endl;
  cout << x->size() << endl;

  delete x;
}
```

Smart Pointers

```
void f() {
  // First, we heap allocate a string
  SPtr<string> x(new string("hi!"));

  cout << *x << endl;
  cout << x->size() << endl;

  // Our string is implicitly deleted
}
```

RAII

I'm a little concerned about how we implemented copying though...

Smart Pointers

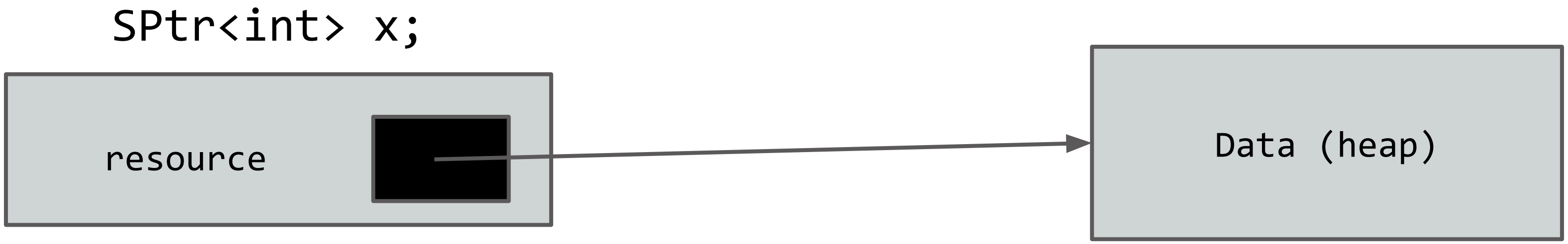
```
// Regular pointers implementation
void f() {
  int *x = new int(4);
  cout << *x << endl;
  int *y = x;
  *y = 8;
  cout << *x << endl;
  delete x;
}
```

Smart Pointers

```
// Will this work given my design?
void f() {
   SPtr<int> x(new int(4));
   cout << *x << endl;
   SPtr<int> y(x);
   *y = 8;
   cout << *x << endl;
}
```

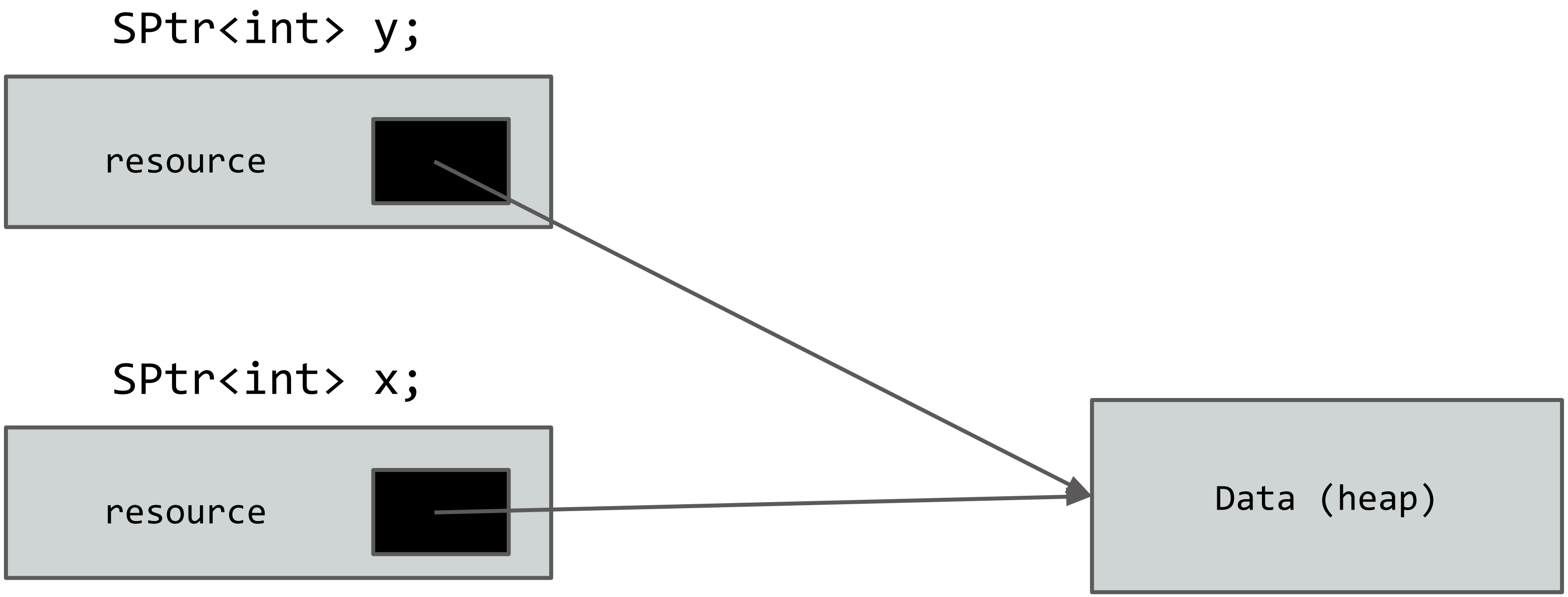
Smart Pointers

First, we set up a smart pointer pointing at our data on the heap



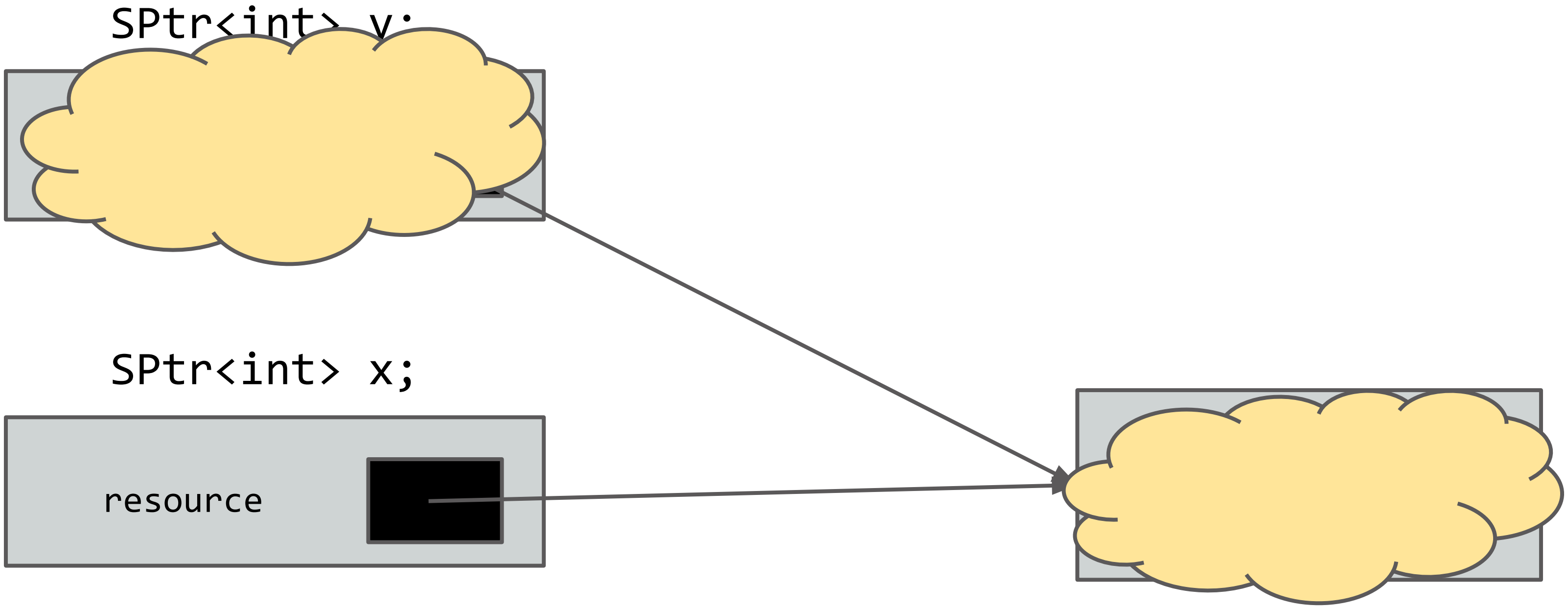
Smart Pointers

We then make a copy of our smart pointer



Smart Pointers

When the function is done, we'll first call the destructor for 'y', implicitly deleting the heap data



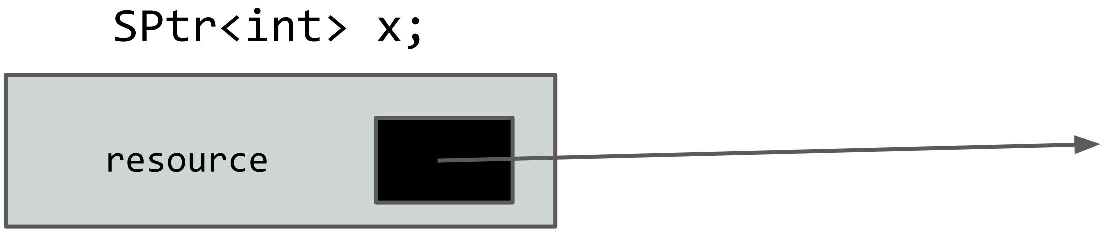
SPtr<int> y;

SPtr<int> x;

resource

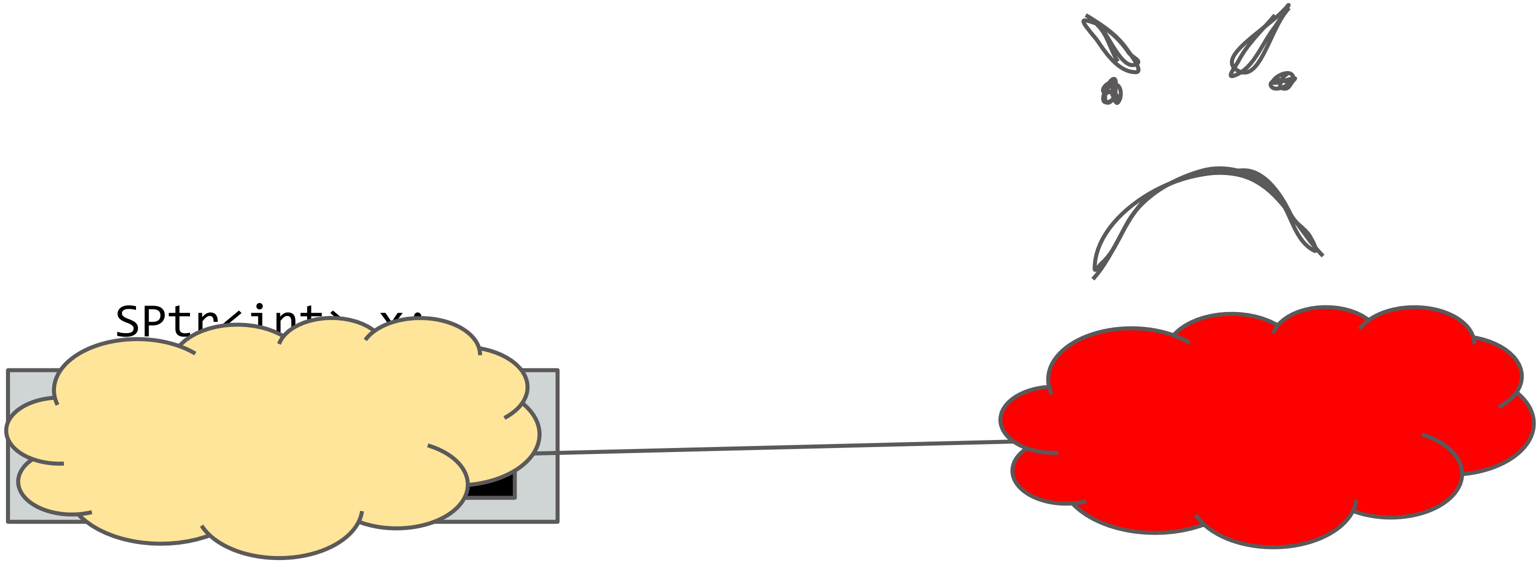
Smart Pointers

This leaves 'x' pointing at deallocated data



Smart Pointers

When we then destroy 'x', we will end up calling delete on the heap data twice!



```
SPtr<int> x;
```

Smart Pointers

You have to be careful when copying an RAII object

You don't want to leave two different objects thinking they exclusively control a resource

Smart Pointers

- Solution #1: Don't allow copying
- This is the approach taken by `std::unique_ptr`
- You **must** pass around a unique_ptr by reference
    - Unless you want to learn more C++11...
- This ensures there is only ever one owner of a pointer

Smart Pointers

- Solution #2: Keep a heap allocated count of references to an object
  - When copying from another object, increment that count
  - When releasing ownership of a pointer, decrement that count, and delete the heap memory if it's zero
- This is the approach taken by `std::shared_ptr`

Smart Pointers

Let's see how we can implement shared and use shared pointers

RAII

To summarize, let me lay out why I think RAII is such a ridiculously cool feature

- RAII isn't magic. We don't need to learn any new syntax
- RAII will help you write less code, do more, and do it in a safer manner
- You've already been using RAII and you don't even know it
- RAII is unique to C++ and a few other languages
  - Take that, Java!